



THE LINUX FOUNDATION



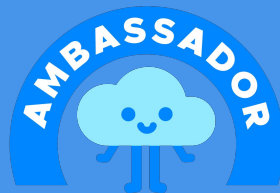
The Next Evolution of Java: ☕ Achieving Hyper Performance and Efficiency in Cloud Native Workloads

Daniel Oh

#OSSUMMIT



Daniel Oh



Java™
Champions



- *Developer Advocate at IBM (Red Hat)*
 - *Cloud Native Runtimes*
 - *AI, GitOps, Security, and Serverless*
- *JAVA Champion*
- *CNCF Ambassador & TAG DevEx co-chair*
- *Microsoft MVP - Java & Developer Tools*
- *Keynote Speaker & Published Author*
- *Living in Boston, MA USA*



@danieloh30

Why Java on Kubernetes Needs Tuning

The Problem: Java is powerful, but Kubernetes exposes its traditional weaknesses:

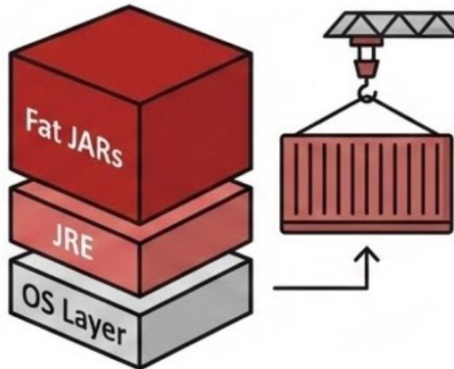
Slow Startup (Cold Start)



Traditional JVM startup:
seconds to tens of seconds.

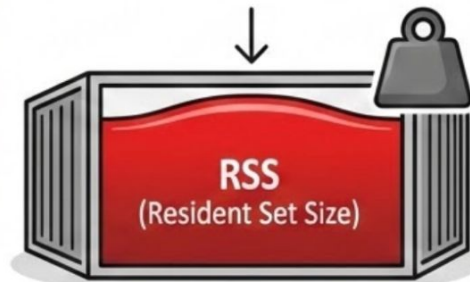
The JVM needs time to warm up
(JIT compilation, class loading).

Large Container Images



Fat JARs + OS layer + JRE =
huge download/deploy times.

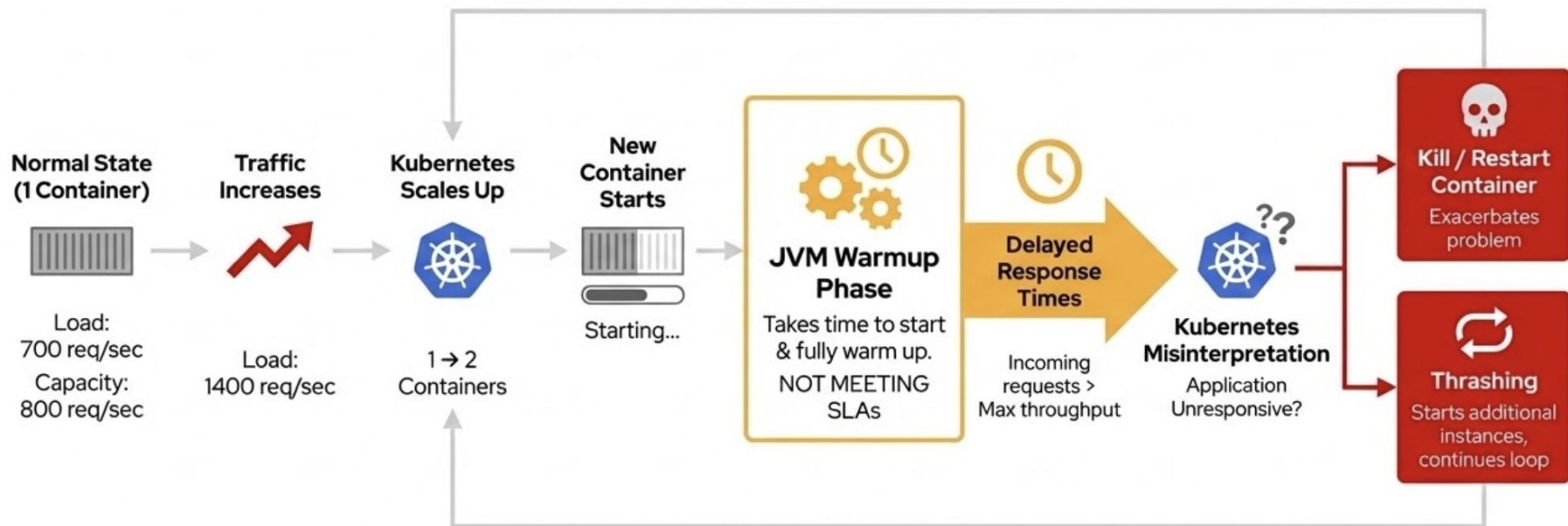
Memory Consumption



Memory overhead (JVM + libraries).
High RSS due to large JRE overhead.
Container resource limits add pressure.

Java wasn't initially Cgroup aware. Cost & efficiency matter at scale.

JVM Scaling in Kubernetes



Customer Story: Vodafone Greece

Spring Boot misbehaved under load, K8s exacerbated.
Solved by moving to Quarkus.



Note on startupProbe:
Useful, but doesn't resolve performance issues.

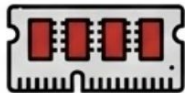
Performance Goals

Success Criteria for Java in Containerized Environments:



Faster Startup (Cold + Warm)

Essential for rapid scaling and serverless workloads. Reduces time to first response.



Reduced Memory Footprint

Lowers infrastructure costs and allows higher density of application instances.



Smaller & Leaner Containers

Faster deployment times, reduced network bandwidth, and smaller attack surface.

Achieve
With

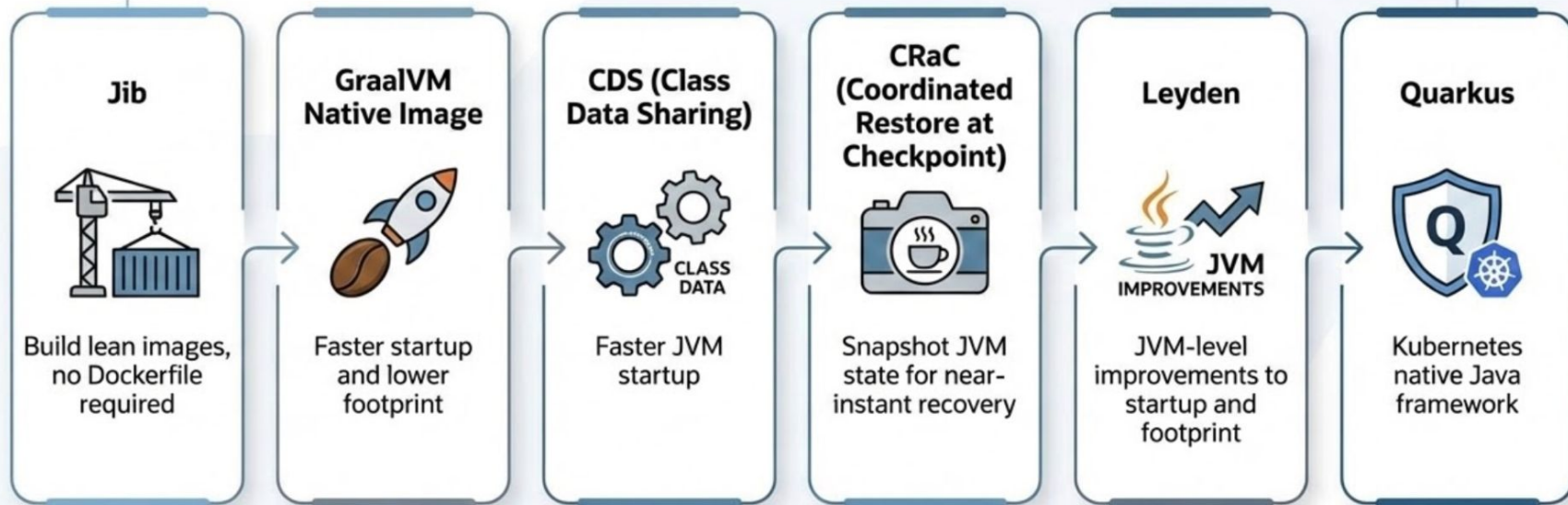


Kubernetes Native Java with Quarkus

A framework designed from the ground up for Kubernetes, optimizing for all the above criteria.

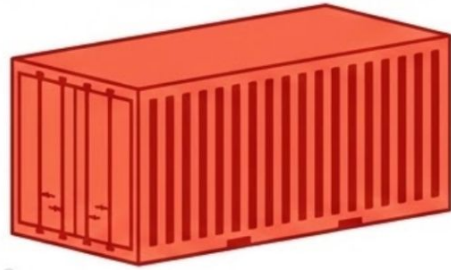
Toolchain Overview

Introduce the tools—each solves different parts of the problem.

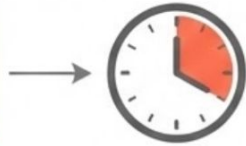


Jib - Lean Container Images

Why Smaller, Layered Images Matter for Faster Kubernetes Rollouts



Traditional Monolithic Image
(Slow Updates)



Jib Layered Image
(Fast Updates)



Smart Layering & Caching

Dependencies and application code are separated. Only the changed layer is rebuilt and pushed, drastically reducing build and deployment times.



Direct Build Integration

Builds directly from Maven or Gradle plugins. No Dockerfile is required, simplifying the project structure.



Daemonless & Secure

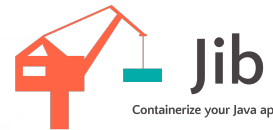
No local Docker daemon is needed. This improves security and is ideal for CI/CD environments without privileged access.



Simple Command

Build and push with a single command:

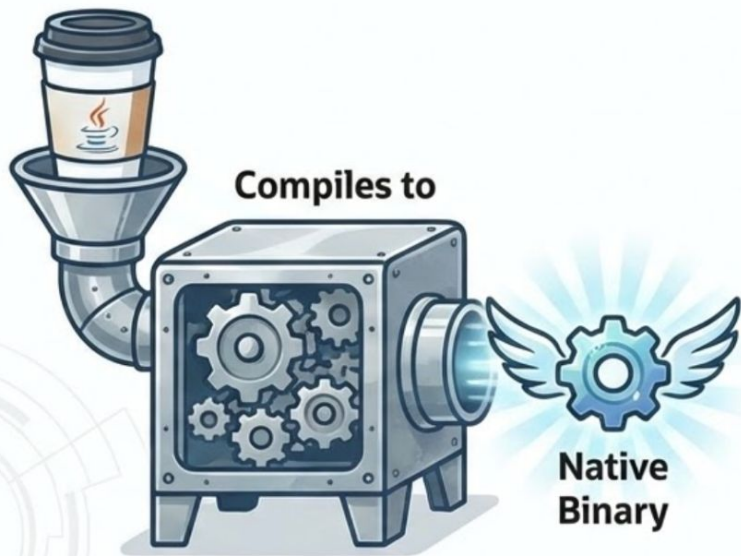
```
mvn compile jib:build
```



Containerize your Java application.

GraalVM Native Image

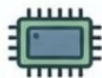
Realistic expectations—native image is not a silver bullet.



+ Benefits



Fast startup
(milliseconds vs.
seconds)



Lower memory
usage

- Trade-offs



Longer build time



Limited
reflection/dynamic
class loading
support

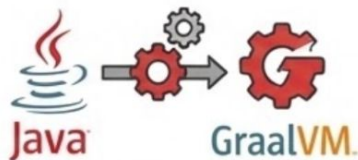
Quarkus – Framework Level Optimizations

Kubernetes Native



Built for Kubernetes & container-first workloads.

GraalVM Support



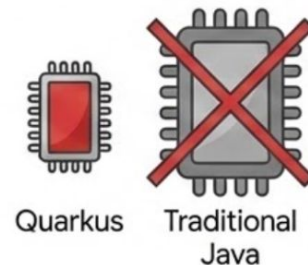
Native image support via GraalVM.

Dev Mode & Rebuilds



Dev mode & fast rebuilds.

Low Memory Footprint



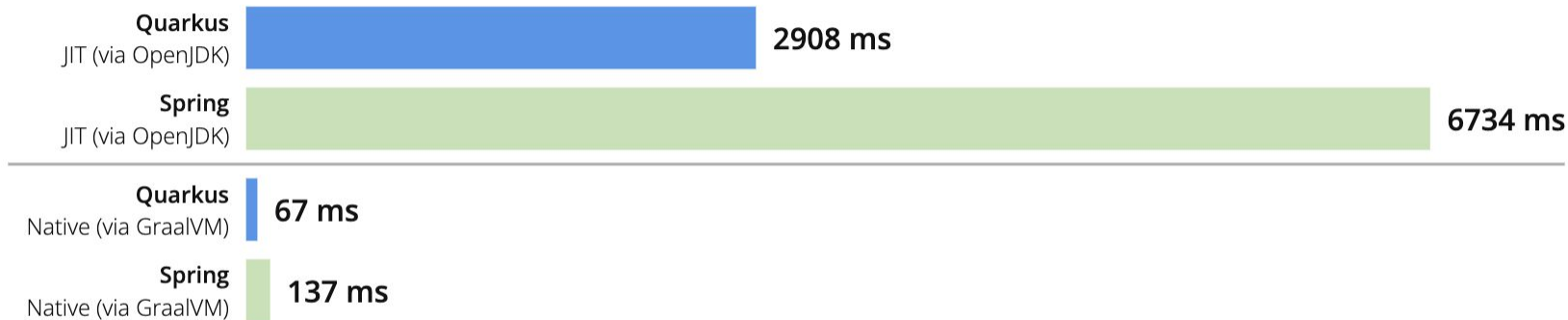
Lower memory footprint than traditional Java stacks.



Quarkus offers unequaled performance I

Boot + First Response Time

(Lower is better)



Quarkus: 3.31.3
Spring: 4.0.2
JVM: 25.0.2-tem
GraalVM: 25.0.2-graalce

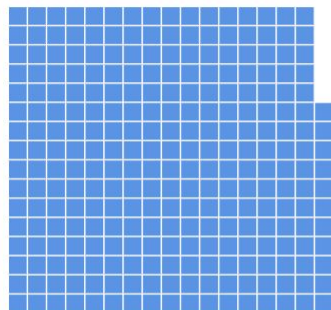
Memory: -Xmx512m -Xms512m
JVM args: -XX:+UseNUMA
CPUS: 4

Source: [quarkusio/spring-quarkus-perf-comparison](https://github.com/quarkusio/spring-quarkus-perf-comparison)
Scenario: tuned
Commit: 152cb4f
Execution date: 2026-02-16 17:12:06

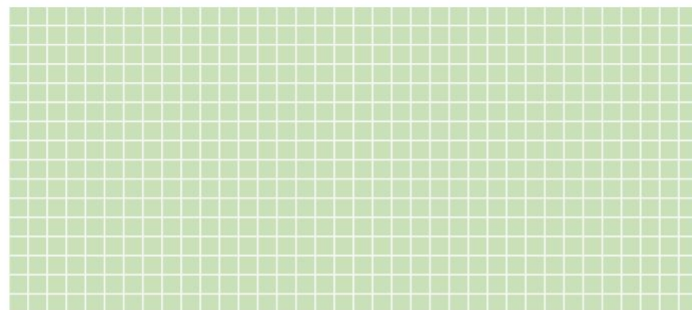
Quarkus offers unequaled performance II

Memory (RSS after first request)

(Smaller is better)



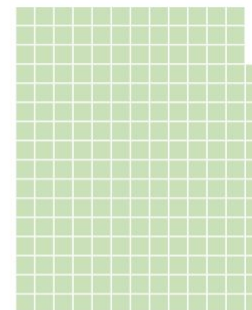
Quarkus
JIT (via OpenJDK)
267 MB



Spring
JIT (via OpenJDK)
576 MB



Quarkus
Native (via GraalVM)
76 MB



Spring
Native (via GraalVM)
205 MB

Quarkus: 3.31.3
Spring: 4.0.2
JVM: 25.0.2-tem
GraalVM: 25.0.2-graalce

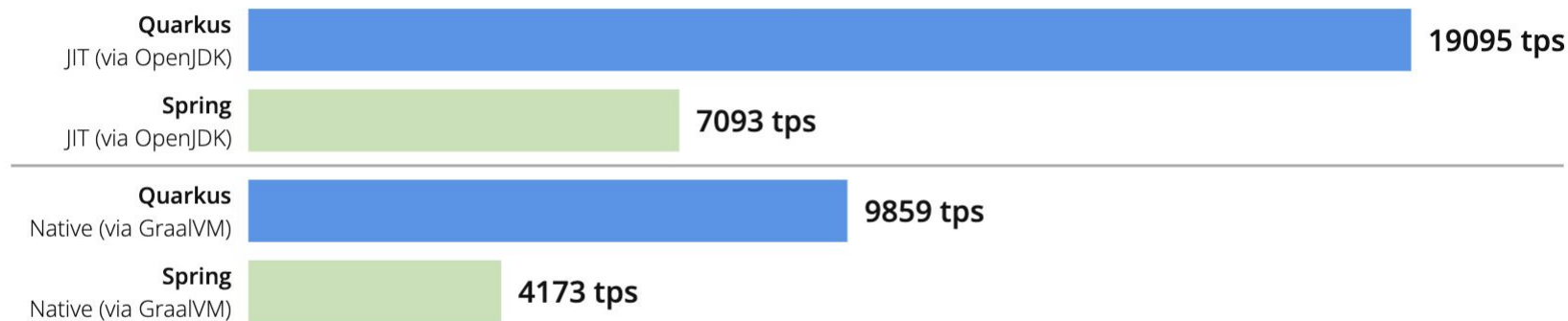
Memory: -Xmx512m -Xms512m
JVM args: -XX:+UseNUMA
CPUS: 4

Source: [quarkusio/spring-quarkus-perf-comparison](https://github.com/quarkusio/spring-quarkus-perf-comparison)
Scenario: tuned
Commit: 152cb4f
Execution date: 2026-02-16 17:12:06

Quarkus offers unequaled performance III


Throughput

(Higher is better)



Quarkus: 3.31.3
Spring: 4.0.2
JVM: 25.0.2-tem
GraalVM: 25.0.2-graalce

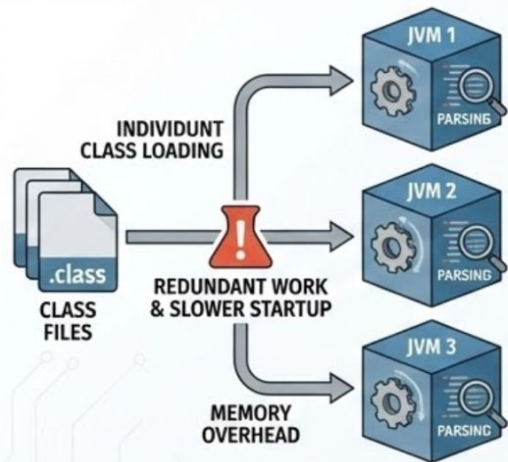
Memory: -Xmx512m -Xms512m
JVM args: -XX:+UseNUMA
CPUS: 4

Source:  quarkusio/spring-quarkus-perf-comparison
Scenario: tuned
Commit: 152cb4f
Execution date: 2026-02-16 17:12:06

CDS – Class Data Sharing

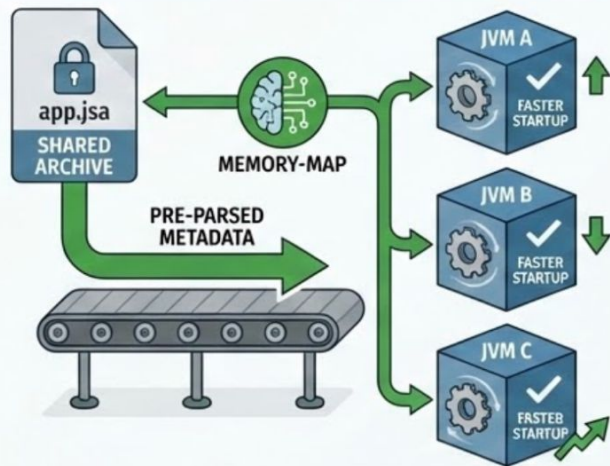
Optimizing Java Startup for Typical Applications

THE PROBLEM: Slow Individual Class Loading



- Traditional Java apps parse class data individually at startup.
- Leads to redundant work and slower initialization.
- Significant memory overhead for each JVM instance.

THE SOLUTION: Class Data Sharing (CDS)



- Preloads class metadata into a shared, memory-mapped archive.
- JVMs skip parsing, enabling much faster startup.
- Reduces memory footprint by sharing read-only data.
- Works seamlessly with OpenJDK and containers.

IMPLEMENTATION: How to Use CDS

1. Create Shared Archive

```
> -XX:ArchiveClassesAtExit=app.jsa
```

Run application once to generate the shared archive file during exit.

2. Use Shared Archive

```
> -XX:SharedArchiveFile=app.jsa
```


Start JVMs using the pre-processed, memory-mapped data.

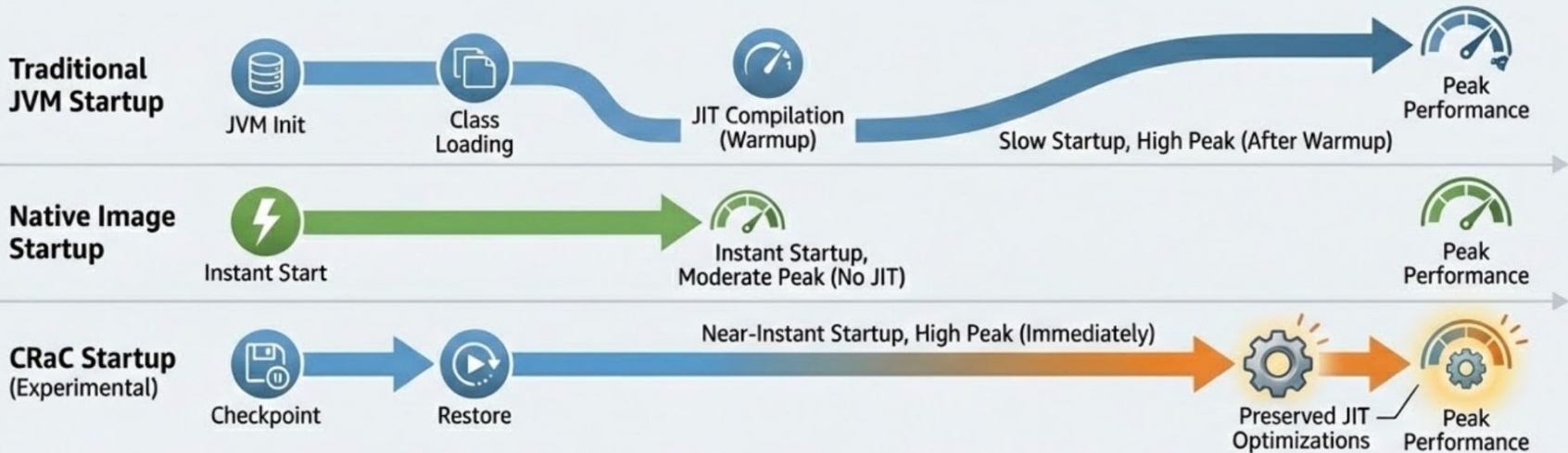
Feature	Traditional	CDS
Class Parsing	Individual	Pre-parsed
Startup Time	Slower	Faster
Memory Usage	Higher	Lower

OpenJDK

<https://openjdk.org/jeps/310>

CRaC – Coordinated Restore at Checkpoint

 **WARNING: EXPERIMENTAL STATUS** Requires a supported JDK. Promising potential, but still under development.



Preserved JIT Optimizations

KEY BENEFIT: Near-Instant Peak Performance

Unlike native images, CRaC restores the application state *including* JIT optimizations, so it runs at peak speed immediately upon restore.

IDEAL USE CASES



Serverless Functions

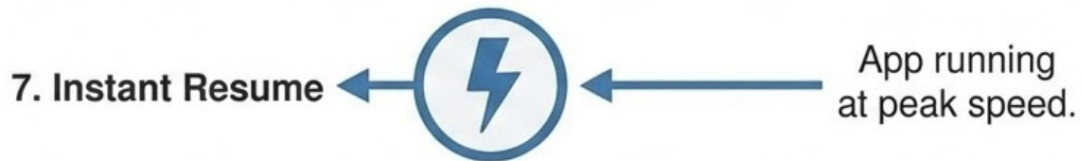
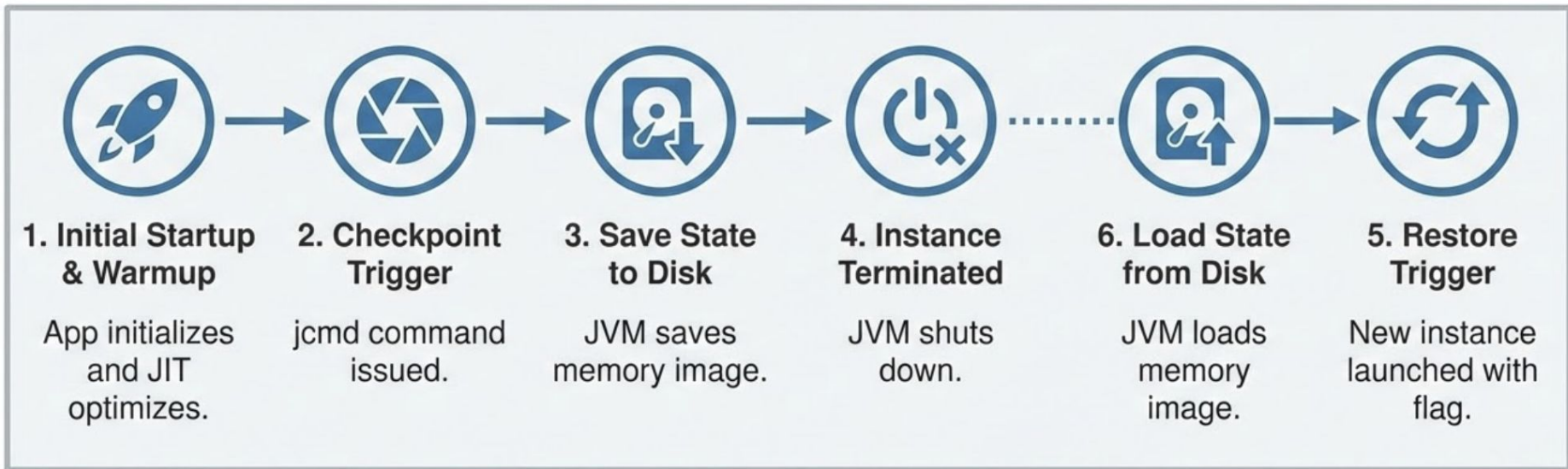


Kubernetes Scaling

CRaC offers a promising future for Java in cloud-native environments, combining the speed of native images with the performance of the JVM.

OpenJDK

CRaC – Coordinated Restore at Checkpoint

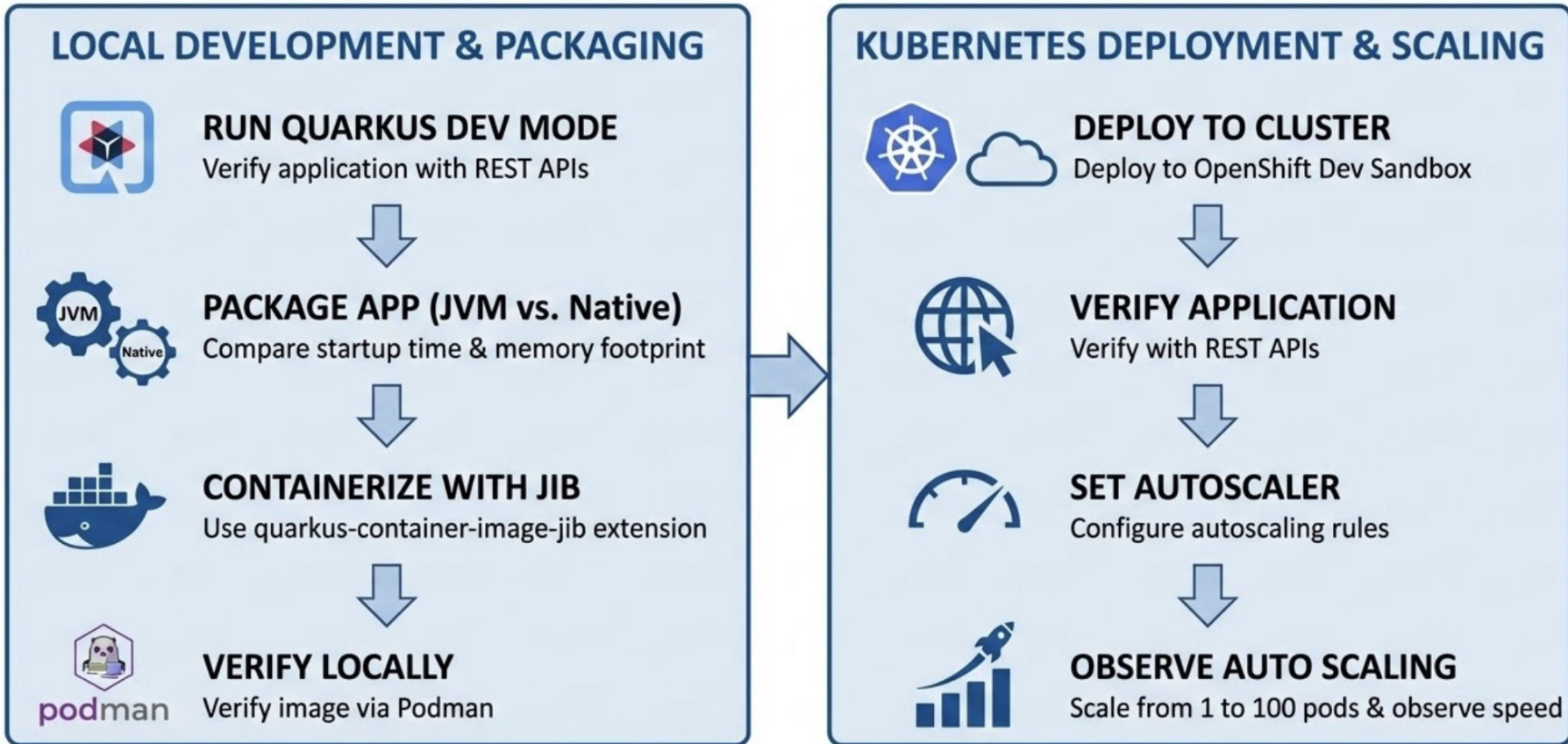


DEMO

Speed up Kube Native Java Development

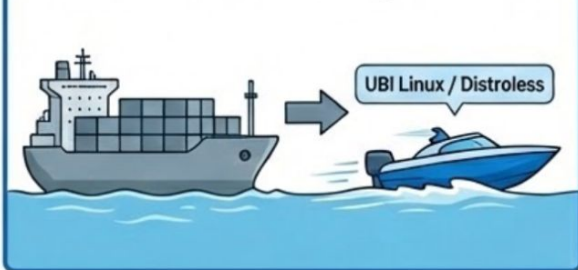


Demo Scenario

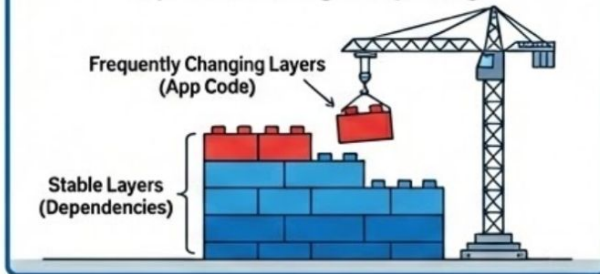


Practical tuning: Reducing Startup Time

Small Base Image



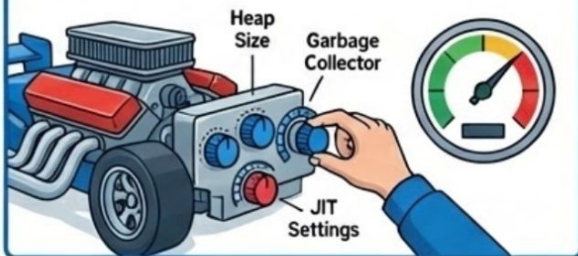
Optimize Image Layering



Reduce Application Footprint



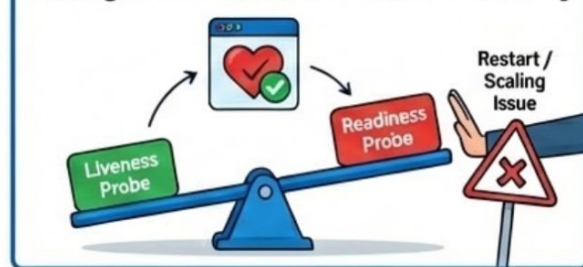
Tune JVM Options



Pre-initialize Resources

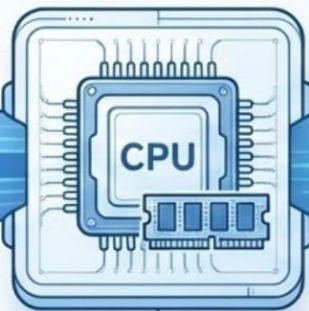


Configure Kubernetes Probes Correctly



Practical tuning: Enhancing Container Efficiency

Resource Management



Set appropriate CPU and RAM limits



Select the right Garbage Collector

Data & Logic Optimization



Optimize database access



Reduce synchronization

Runtime Environment



Upgrade Java versions



Consider Kubernetes overhead

Visibility & Analysis



Monitor and analyze resource usage



Utilize profiling tools

Key Takeaways



**Java can be
Kubernetes-
friendly**



**Demo showed
measurable
improvements**



**Quarkus +
Kubernetes
(OpenShift) allows
you to put all tools
together natively**



THE LINUX FOUNDATION

OS OPEN SOURCE SUMMIT INDIA

